Understanding Code

by Kwazy Webbit

Introduction

There is no single layer of abstraction to executable code. It has many, the lowest of which is binary.
It is important to understand that binary can represent anything. Executable code and data are, at the lowest level, the exact same thing: a collection of 1's and 0's. You can try to run data as code, but most likely that will just cause a crash. Trying to use executable code as, for example, picture data will also be either invalid or, at best, just random. This is because there is a structure to both of them which allows them to become more than just binary. To be useful, you need something that understands this structure and will interpret it in the proper way.
As a more concrete (non-binary) example, one could have four numbers:

```
112,43,149,184
```

They could mean pretty much anything. If I were to tell you it was a line for example, one could imagine it being a line in 2 dimensions, starting at coordinates `(112,43)`, and ending in `(149,184)`. However, if I were to tell you it was a square, you could think of it as a square with those coordinates as top-left and bottom-right points.
It can be anything, it all depends on your interpretation. The problem is, how do we make sense of it all? How does the computer know what to do with what? How can WE know what it really does? In this essay, I will not go into understanding data, since data structures are too diverse (think alone of the image formats you've seen). Each file format has a different structure. Programs use the file extension as a hint for the structure to expect.
Instead, I will focus on executable code, specifically that for the x86 processor. I will start at binary, and eventually end up in C[*].

---

[*] I chose the C programming language because it is very close to regular mathematical notation, and simple expressions are fairly easy to read even for non-programmers. If you do have trouble understanding C, there is a plethora of information to be found on the web

Binary to Hexadecimal

As mentioned before, the lowest level of information (in a computing environment) is binary. Code, as the computer sees it, is an endless row of 1's and 0's. It is nearly impossible for humans to follow what is happening by seeing it. If you are interested in how the circuits in your CPU work, I suggest getting some electronics books. I do not know enough about it to explain in detail how it works (though I have seen it work on much simpler processors). For purposes of explanation, binary is a clumsy format, as the amount of binary digits is too large for us to easily oversee. That is why we never normally edit anything in binary, but instead go to a direct translation, known as hexadecimal format. It is just a numbering format. Just as a numerical representation of binary has two digits: 0 and 1, and decimal has 10 (0, 1, 2, 3, 4, 5, 6, 7, 8, 9), hexadecimal has 16: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. You may wonder why this format is chosen over the decimal system which we are all used to working with, and is thus much more intuitive for us humans. The answer is simple. It is because underneath it all the numbers are still binary, being translated. Using 4 bits at a time you can make exactly 16 different values, from 0 to 15. In hexadecimal, that is from 0 to F. This makes it a very practical system to write down 4 bits with one character. I've included a quick lookup table, if you aren't familiar with it already.

Binary	Decimal	Hexadecimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F
10000	16	10

As you can see, at binary `10000`, the hexadecimal value (`10`) still makes sense, since the first digit (`1`) can still represent `0001`, while the second digit (`0`) represents the `0000`, resulting in `00010000`, which matches the binary. The decimal however is now `16`, which is not as obvious to convert anymore.

Hexadecimal to Assembly Code

Using hexadecimal notation we have a shorthand for writing down binary code, giving us more overview. It still makes no sense to a human though, because essentially it is still just a lot of numbers. Following is a sample of code in hexadecimal format:

```
83EC20535657FF158C40400033DBA39881400053
```

As said, this is just a shorthand for the binary digits it represents. That means it doesn't give any explanation to what it does, but it is a lot shorter than the binary representation: The hexadecimal representation is 40 characters, while the binary would be 160 (Since every hexadecimal digit represents 4 bits)

The code above is not one big instruction[*]. It is, in reality, several small ones. On some processors, every instruction has a certain size (e.g. 2 bytes) so you can easily chop code up into parts of that size to get the different instructions (assuming you have a valid starting position). The x86 processor is a little more complex (it is a CISC[†] architecture after all), and has different sized instructions. You might now wonder how we can ever split up different instructions this way. The idea is, you take the first byte, look at its value, and that byte will tell you how to proceed. Several things could happen:

- It could be a single byte instruction: e.g. `90h`[‡] is the 'NOP' instruction (No OPeration) and is only 1 byte in size.
- The instruction is not yet complete: e.g. Instructions starting with `0Fh` need more bytes to fully define their function.
- The instruction is defined by a single byte, but it needs parameters: e.g. `8Bh` moves one register into another. The byte following `8Bh` will describe where it moves from, and where it moves to.
- The instruction is not yet complete <u>and</u> it needs parameters.

Because we will need to know what an instruction is in order to split them up anyway, we will combine the process of splitting up the different instructions with translating them to a human-readable equivalent. This 'human-readable equivalent' is known as 'assembly language', often abbreviated to ASM. The process of translating a program from raw code to ASM, is known as 'disassembling' (lit. 'taking apart'). It takes some skill to read ASM. However, since every instruction in ASM performs a fairly trivial task (even on a

---

[*] The term 'instruction' here refers to the actual code bytes, while the term 'operation' refers to the task that instruction performs.

[†] Complex Instruction Set Computer, that means it has a lot of different instructions which do a lot of different detailed things. Its counterpart is the RISC (Reduced Instruction Set Computer) where only a few instructions exist, doing simple tasks. This allows lower complexity, and single instructions are generally executed faster. However, since less is done in a single instruction, it has been long debated which solution is better.

[‡] Hexadecimal values are generally indicated by appending a 'h'. In C notation, they are represented by preceding the value with '0x', for example '0x90' means 90h.

CISC processor), they are easy to understand by themselves. It takes some experience however, to keep an overview of what non-trivial action is being done, more on this later. First, we are going to look at the separate instructions.

Since there is no clear system to see what operation a hexadecimal code performs (it is basically a matter of looking it up in a reference, and writing it down), it is a rather tedious job. However, as it is important to understand how this works, I will demonstrate using the example above.
Let's take another look at the hexadecimal code:

```
83EC20535657FF158C40400033DBA39881400053
```

We will assume the first byte is a valid starting point (and not halfway through an instruction, because this would ruin our disassembly process[*]) and go from there.
We take the first byte, which is `83h`, and we'll take a manual to look it up. I used the table in Appendix A1[†] to look it up. This says it requires another byte to describe the full operation, and that this byte should be in the form of a 'mod R/M' byte. To see what the full operation is to use the information from this byte and look under "group #1" in Appendix A2. In this case, the byte is `ECh`. A mod R/M byte consist of 3 bitfields:

Bit :	7	6	5	4	3	2	1	0
Meaning :	mod		reg			R/M		

To separate these bitfields, we have to go back to binary, which gives:

```
EC = 1110 1100 = 11 101 100
```

Using Appendix A2 we see that for the bitfields matching xx101xxx, the operation is `SUB`. The other two bitfields describe the first operand of the `SUB` operation. Looking at Appendix B, we find that `11` means that it uses a register directly, and `100` means that that register is `ESP`. Using the original description of Appendix A1, we have one more operand to fill, the 'Ib' (Input byte). Quite simply, the next byte is to be used, which is the 3rd byte (`20h`).
Putting all this together, we find the first ASM instruction:

```
83EC20	SUB ESP, 20
```

Which lets us continue to decode the next instruction (starting with `53h`).
Let's do one more. Looking up `53h` (in Appendix A1) shows it is a single byte instruction with no parameters:

```
PUSH rBX (= PUSH EBX)
```

---

[*] Some programs purposely use this 'starting point'-problem to confuse disassemblers, to prevent outsiders looking at how their program works. This technique is known as 'obfuscation'.
[†] The tables in Appendix A are taken from http://www.sandpile.org

So now we have translated the first 4 bytes into their ASM equivalents:

```
83EC20        SUB ESP, 20
53            PUSH EBX
```

As you probably realized by now, disassembling takes quite long to do manually. Luckily, there are plenty of ready made tools (aptly called 'disassemblers') to perform this process for us (e.g. HIEW).
Using HIEW, the hexadecimal example is translated to ASM as follows:

```
83EC20                        sub         esp,020
53                            push        ebx
56                            push        esi
57                            push        edi
FF158C404000                  call        d,[0040408C]
33DB                          xor         ebx,ebx
A398814000                    mov         [00408198],eax
53                            push        ebx
```

Some programs are a bit more clever though, trying to understand the flow of a program. For example, it could look at the addresses used and see which ones point to a string, or analyze the flow of the program (following jumps). These more advanced disassemblers include IDA and WDasm. Using IDA, the most advanced disassembler available at the time of writing, the result is:

```
sub     esp, 20h
push    ebx
push    esi
push    edi
call    ds:GetProcessHeap
xor     ebx, ebx
mov     hHeap, eax
push    ebx             ; lpModuleName
```

As you can see, IDA has done some more analysis. Here, it has figured out where the call is going, and it understands that the return value from that Windows function (`GetProcessHeap`) is a handle to a heap, so it has appropriately renamed the variable to `hHeap`. In this example there was little IDA could do, but usually it gives quite a lot more information than a less specialized program such as HIEW.
These advanced features save us a lot of work looking into everything manually, and gives a good starting point for further analysis of the program. In ASM, we can see what the code is doing one little step at a time, but to make this useful we need a bigger picture. A higher level of abstraction minimizes explanations of HOW things are happening, thus leaving the focus on WHAT is happening. A language like C gives us that overview.

Assembly code to C

Now that we have the ASM code, it is understandable for humans what the program is doing. However, since every ASM instruction only performs a trivial task, it is hard to see what non-trivial function a program is performing. Let's see an ASM listing as created by HIEW:

```
.004122F0: 55                     push      ebp
.004122F1: 8BEC                   mov       ebp,esp
.004122F3: 83EC48                 sub       esp,048 ;"H"
.004122F6: 53                     push      ebx
.004122F7: 56                     push      esi
.004122F8: 57                     push      edi
.004122F9: C745F800000000         mov       d,[ebp][-08],000000000 ;"
.00412300: EB09                   jmps     .00041230B  -----↓ (1)
.00412302: 8B45F8                 mov       eax,[ebp][-08]
.00412305: 83C001                 add       eax,001 ;"☺"
.00412308: 8945F8                 mov       [ebp][-08],eax
.0041230B: 8B4508                 mov       eax,[ebp][08]
.0041230E: 50                     push      eax
.0041230F: FF1584A34300           call      lstrlenA ;KERNEL32.dll
.00412315: 3945F8                 cmp       [ebp][-08],eax
.00412318: 7D2E                   jge      .000412348  -----↓ (2)
.0041231A: 8B4508                 mov       eax,[ebp][08]
.0041231D: 0345F8                 add       eax,[ebp][-08]
.00412320: 8A08                   mov       cl,[eax]
.00412322: 884DFF                 mov       [ebp][-01],cl
.00412325: 0FB645FF               movzx     eax,b,[ebp][-01]
.00412329: 83F861                 cmp       eax,061 ;"a"
.0041232C: 7C18                   jl       .000412346  -----↓ (1)
.0041232E: 0FB645FF               movzx     eax,b,[ebp][-01]
.00412332: 83F87A                 cmp       eax,07A ;"z"
.00412335: 7F0F                   jg       .000412346  -----↓ (2)
.00412337: 0FB645FF               movzx     eax,b,[ebp][-01]
.0041233B: 83E820                 sub       eax,020 ;" "
.0041233E: 8B4D08                 mov       ecx,[ebp][08]
.00412341: 034DF8                 add       ecx,[ebp][-08]
.00412344: 8801                   mov       [ecx],al
.00412346: EBBA                   jmps     .000412302  -----↑ (3)
.00412348: 5F                     pop       edi
.00412349: 5E                     pop       esi
.0041234A: 5B                     pop       ebx
.0041234B: 8BE5                   mov       esp,ebp
.0041234D: 5D                     pop       ebp
.0041234E: C3                     retn
```

As you can see, ASM uses a lot of simple instructions to work together and ultimately perform a useful task. We'll start at the first instruction and work down, trying to keep an overview of what is going on, using a 'Pseudo-C'* notation, and eventually translating to proper C code.

* Called 'Pseudo-C' because, even though it follows the general structure of C in terms of operators, it is a literal translation of the ASM code, and thus still uses registers directly.

Here are the first few lines:

```
.004122F0: 55                       push     ebp
.004122F1: 8BEC                     mov      ebp,esp
.004122F3: 83EC48                   sub      esp,048 ;"H"
.004122F6: 53                       push     ebx
.004122F7: 56                       push     esi
.004122F8: 57                       push     edi
```

The first two operations create what is known as a 'stackframe'. This is essentially a 'local' stack inside the function, where extra room can be reserved for local variables. This is done simply by lowering the stack pointer a bit further, for as many bytes as are necessary for the local variables.
One of the main advantages of a stackframe is that the `EBP` register can be used as a fixed point to reference variables (above `EBP` are the parameters, and below it are the local variables).
Note that the stackpointers (`ESP` and `EBP`) have to be restored before leaving the function, to avoid stack corruption.

```
.004122F0: 55                       push     ebp
.004122F1: 8BEC                     mov      ebp,esp
.004122F3: 83EC48                   sub      esp,048 ;"H"
```

This code creates such a stack frame, and creates 48h bytes of space for local variables.

Windows requires that a few registers (besides `ESP` and `EBP`) are preserved during a callback function, namely `EBX`, `ESI`, and `EDI`. These are stored safely on the (local) stack, ready to be restored right before leaving the function. This allows for free use of these registers while inside the function.
It seems most practical to look at the last few instructions, since we now already know several tasks that need to be performed there. And looking, we indeed find exactly what we expected:

```
.00412348: 5F                       pop      edi
.00412349: 5E                       pop      esi
.0041234A: 5B                       pop      ebx
.0041234B: 8BE5                     mov      esp,ebp
.0041234D: 5D                       pop      ebp
.0041234E: C3                       retn
```

First, the 3 registers are restored from our (local) stack. Then the stack is restored to its state when the function was called, and the function returns. Note that we could not have restored the stack before the 3 registers, because the registers were stored on our local stack. Converting all this to C is very easy. We know now that it is probably a function, because of it's stackframe, storing/restoring registers, as well as its `retn` (return from function) at the end:

```
void SomeFunction()
{
	//…code…
}
```

I've assumed for now that this is a void function, because there is no change in EAX before the return. This does not mean EAX is never changed. But for now, we will assume the value in EAX is ignored.

Now we will proceed further into the body of this function:

```
.004122F9: C745F800000000              mov        d,[ebp][-08],000000000 ;"
.00412300: EB09                        jmps      .00041230B  -----↓ (1)
.00412302: 8B45F8                      mov        eax,[ebp][-08]
.00412305: 83C001                      add        eax,001 ;"☺"
.00412308: 8945F8                      mov        [ebp][-08],eax
.0041230B: 8B4508                      mov        eax,[ebp][08]
.0041230E: 50                          push       eax
.0041230F: FF1584A34300                call       lstrlenA ;KERNEL32.dll
.00412315: 3945F8                      cmp        [ebp][-08],eax
.00412318: 7D2E                        jge       .000412348  -----↓ (2)
```

We notice a value being referenced a lot:

```
d,[ebp][-08] == dword ptr[ebp-08] (in another notation)
```

Since it is below our EBP it is on our local stack, so the function is storing a local variable there. We know that it is DWORD size (because it's being read using a dword ptr), and that it's probably a signed value (because it's being compared to the result of lstrlenA, which is a signed int). On the win32 platform, the standard signed dword size value in C is the (signed) int. Let's rename it to `int_local1` for easier reading (I also removed the hexadecimal representation of the code, and the less helpful comments):

```
.004122F9: mov        int_local1, 000000000
.00412300: jmps      .00041230B  -----↓ (1)
.00412302: mov        eax, int_local1
.00412305: add        eax,001
.00412308: mov        int_local1, eax
.0041230B: mov        eax,[ebp][08]
.0041230E: push       eax
.0041230F: call       lstrlenA ;KERNEL32.dll
.00412315: cmp        int_local1,eax
.00412318: jge       .000412348  -----↓ (2)
```

Be careful here. Do not confuse `[ebp][08]` with `[ebp][-08]`. Even though they look alike, they are different addresses. The variable at `[ebp][08]` is *always* (assuming a normal stackframe) the first parameter passed to our function. We will thus (for the time being) rename that value to `dw_param1`. Now that we have identified a local variable, and cleaned things up a little, we will make a start at converting to Pseudo-C:

```
      int_local1 = 0;
      goto label_41230B;
      eax = int_local1;
      eax = eax + 1;
      int_local1 = eax;
label_41230B:
      eax = dw_param1;
      eax = lstrlenA(eax); //lstrlenA returns its result in eax
```

```
	if( int_local1 >= eax)
		goto label_412348;
```

Rather strange looking code, but it’s a start. Let’s be a little bit less literal about it, and use our brain. Looking at the 3 lines:

```
	eax = int_local1;
	eax = eax + 1;
	int_local1 = eax;
```

We see that it is really a very simple instruction, that could be simplified into a mere:

```
int_local1++;
```

The only difference between these two representations though, is that EAX no longer has the same value after the new representation. We should take care in doing so, because the value in EAX might be used afterwards.
In this case, the next line is:

```
	eax = dw_param1;
```

Which means we can freely replace the instruction, since EAX gets overwritten before being read anyway. The next part:

```
	eax = dw_param1;
	eax = lstrlenA(eax); // lstrlenA returns its result in eax
	if( int_local1 >= eax)
		goto label_412348;
```

could also be made a lot easier to view, because you can combine a lot of instructions in C. We can do it as follows:

```
	if( int_local1 >= lstrlenA(dw_param1) )
		goto label_412348;
```

Again, we should now look at if EAX is used afterwards, so that we don’t miss another location where this value was being used. On the very next line, however, EAX is overwritten, so we are free to make this change. Because we know lstrlenA is expecting a pointer to a string, we will rename the parameter now to `pString` to represent this, giving us a total code of:

```
	int_local1 = 0;
	goto label_41230B;
	int_local1++;
label_41230B:
	if( int_local1 >= lstrlenA(pString))
		goto label_412348;
```

Looking for further references to this section, we find the line

```
.00412346: EBBA                          jmps       .000412302  -----↑ (3)
```

jumping to the line with `int_local1++;`, which makes it all appear to be big loop. If you're familiar with C coding, you might already have figured out what structure we are looking at here. It seems to have the functionality of a 'for'-loop. Let's try to make a for-loop that mimics this behavior (while also renaming `int_local1` to `i`). Rewriting the C code, we end up with:

```
for(i = 0; i < lstrlenA(pString); i++)
{
 //…rest of code…
}
```

It's slowly beginning to make sense. We now know it is a function that goes through a for-loop, ranging from 0 to the length of the string it gets as its (first) parameter. Now we need to know what it does inside the loop.
The code was:

```
.0041231A: mov         eax, pString
.0041231D: add         eax,i
.00412320: mov         cl,[eax]
.00412322: mov         [ebp][-01],cl
.00412325: movzx       eax,b,[ebp][-01]
.00412329: cmp         eax,061 ;"a"
.0041232C: jl         .000412346  -----↓ (1)
.0041232E: movzx       eax,b,[ebp][-01]
.00412332: cmp         eax,07A ;"z"
.00412335: jg         .000412346  -----↓ (2)
.00412337: movzx       eax,b,[ebp][-01]
.0041233B: sub         eax,020 ;" "
.0041233E: mov         ecx, pString
.00412341: add         ecx, i
.00412344: mov         [ecx],al
```

Here, we find another local variable in use. It appears to be of the `unsigned char` type, because it is byte size (referenced by using a byte ptr), and is used as unsigned (by the movzx instructions). In Pseudo-C, we now have:

```
eax = pString;
eax = eax + i;
cl = *(eax);
ch_local2 = cl;

eax = (DWORD) ch_local2;
if(eax < 0x61) // "a"
       goto label_412346;

eax = (DWORD) ch_local2;
if(eax > 0x7A) // "z"
       goto label_412346;

eax = (DWORD) ch_local2;
eax = eax – 0x20;
ecx = pString;
ecx = ecx + i;
*(ecx) = al;
```

Let’s make this code a bit more clever, and thus shorter, renaming the character to `c` for shortness, as well as assuming its using the char as a character and not as a byte sized number:

```
c = pString[i];

if((c < ‘a’) || (c > ‘z’))
	goto label_412346;

pString[i] = c-0x20;
```

We notice that the address `412346h` is simply the end of the loop, so we can either replace the ‘`goto label_412346`’ with a ‘`continue;`’, or we can invert the conditional jumps. I chose the latter, because it seemed like a more natural way to describe the condition, as you will see later. Inverting the condition might require some explanation:
When the program goes to the end of the loop if `(c<‘a’)||(c>‘z’)`, then it DOESN’T go to the end of the loop, if `(c>=’a’)&&(c<=’z’)`, which allows an if construction as follows:

```
c = pString[i];

if((c >= ‘a’) && (c <= ‘z’))
	pString[i] = ch_local2-0x20;

//…end of loop
```

This makes it look MUCH clearer. We can now begin to understand what this code is doing. Let’s put all of the code we have together.

```
void SomeFunction(char* pString)
{
	int i;			//Local variables have to be declared
	unsigned char c;		//at the start of the function.

	for(i = 0; i < lstrlenA(pString); i++)
	{
		c = pString[i];

		if((c >= ‘a’) && (c <= ‘z’))
			pString[i] = c-0x20;
	}
}
```

A great deal shorter than the ASM code we started from. Now that we’ve converted all this back to C, we should be able to figure out the task it performs.
It takes every character in the string it gets, and if that character is between ‘a’ and ‘z’ (so, if it is a lowercase alphabetic character), it subtracts 20h.This is exactly the difference between the uppercase and lowercase characters. So what this function does is ‘convert a string to uppercase’, and should be renamed as `ToUppercase`.
In this manner all code can slowly be converted, though some structures are harder to identify than others.

Conclusion

A normal engineering process goes from the source code (C) to a binary format (.exe), while what I have described in this document goes entirely the other way. That is the reason this process is called Reverse Engineering. We have seen this is not impossible to do. With tools however, the task can be simplified a lot. The main tool one would use for this kind of thing is IDA. It is both flexible and powerful, and even for the translation back to (pseudo) C code there are plugins under development[*]. To create proper C code from an ASM or Pseudo-C listing is a task not to be underestimated. It is quite hard to recognize high level structures at first. A good exercise is to write your own program in MSVC++, and debug it with the disassembly view on. This gives your C code along with the ASM code it represents, which will give you a good understanding of how the two relate to one another.

As with most things, practice makes perfect.

Kwazy Webbit

Webbithole: http://its.mine.nu/
RETeam: http://www.reteam.org/

Special thanks go to DEATH, for proofreading this essay and being a perfectionist :-)

[*] IDA (http://www.datarescue.com/) combined with Lantern (http://www.xopesystems.com/lantern/) forms a tool that automates almost the entire process I described in this document. The only thing it doesn't do is create actual C code, since that requires a lot of understanding and recognition of structures. It does create the Pseudo-C code I have used throughout the 'Assembly Code to C' chapter. See their site for details.

Appendix A1

Key to decoding single-byte instructions (`#group` references Appendix A2)

IA-32 architecture
one byte opcodes

xxh	x0h	x1h	x2h	x3h	x4h	x5h	x6h	x7h
0xh	ADD Eb,Gb	ADD Ev,Gv	ADD Gb,Eb	ADD Gv,Ev	ADD AL,Ib	ADD rAX,Iz	PUSH ES	POP ES
1xh	ADC Eb,Gb	ADC Ev,Gv	ADC Gb,Eb	ADC Gv,Ev	ADC AL,Ib	ADC rAX,Iz	PUSH SS	POP SS
2xh	AND Eb,Gb	AND Ev,Gv	AND Gb,Eb	AND Gv,Ev	AND AL,Ib	AND rAX,Iz	ES:	DAA
3xh	XOR Eb,Gb	XOR Ev,Gv	XOR Gb,Eb	XOR Gv,Ev	XOR AL,Ib	XOR rAX,Iz	SS:	AAA
4xh	INC eAX	INC eCX	INC eDX	INC eBX	INC eSP	INC eBP	INC eSI	INC eDI
5xh	PUSH rAX	PUSH rCX	PUSH rDX	PUSH rBX	PUSH rSP	PUSH rBP	PUSH rSI	PUSH rDI
6xh	PUSHA PUSHAD (80186+)	POPA POPAD (80186+)	BOUND Gv,Ma (80186+)	ARPL Ew,Gw (80286+)	FS: (80386+)	GS: (80386+)	OPSIZE: (80386+)	ADSIZE: (80386+)
7xh	JO Jb	JNO Jb	JB Jb	JNB Jb	JZ Jb	JNZ Jb	JBE Jb	JNBE Jb
8xh	group #1 Eb,Ib	group #1 Ev,Iz	group #1* Eb,Ib	group #1 Ev,Ib	TEST Eb,Gb	TEST Ev,Gv	XCHG Eb,Gb	XCHG Ev,Gv
9xh	NOP PAUSE (F3h) (see CPUID)	XCHG rCX,rAX	XCHG rDX,rAX	XCHG rBX,rAX	XCHG rSP,rAX	XCHG rBP,rAX	XCHG rSI,rAX	XCHG rDI,rAX
Axh	MOV AL,Ob	MOV rAX,Ov	MOV Ob,AL	MOV Ov,rAX	MOVS Yb,Xb	MOVS Yv,Xv	CMPS Yb,Xb	CMPS Yv,Xv
Bxh	MOV AL,Ib	MOV CL,Ib	MOV DL,Ib	MOV BL,Ib	MOV AH,Ib	MOV CH,Ib	MOV DH,Ib	MOV BH,Ib
Cxh	group #2 Eb,Ib (80186+)	group #2 Ev,Ib (80186+)	RET near Iw	RET near	LES Gz,Mp	LDS Gz,Mp	group #12 Eb,Ib	group #12 Ev,Iz
Dxh	group #2 Eb,1	group #2 Ev,1	group #2 Eb,CL	group #2 Ev,CL	AAM Ib	AAD Ib	SALC SETALC	XLAT
Exh	LOOPNE LOOPNZ Jb	LOOPE LOOPZ Jb	LOOP Jb	JCXZ JECX Jb	IN AL,Ib	IN eAX,Ib	OUT Ib,AL	OUT Ib,eAX
Fxh	LOCK:	INT1 (ICEBP) (80386+)	REPNE:	REP: REPE:	HLT	CMC	group #3 Eb	group #3 Ev

xxh	x8h	x9h	xAh	xBh	xCh	xDh	xEh	xFh
0xh	OR Eb,Gb	OR Ev,Gv	OR Gb,Eb	OR Gv,Ev	OR AL,Ib	OR rAX,Iz	PUSH CS	two byte opcodes (80286+)
1xh	SBB Eb,Gb	SBB Ev,Gv	SBB Gb,Eb	SBB Gv,Ev	SBB AL,Ib	SBB rAX,Iz	PUSH DS	POP DS
2xh	SUB Eb,Gb	SUB Ev,Gv	SUB Gb,Eb	SUB Gv,Ev	SUB AL,Ib	SUB rAX,Iz	CS: Hint Not Taken for Jcc (P4+)	DAS
3xh	CMP Eb,Gb	CMP Ev,Gv	CMP Gb,Eb	CMP Gv,Ev	CMP AL,Ib	CMP rAX,Iz	DS: Hint Taken for Jcc (P4+)	AAS
4xh	DEC eAX	DEC eCX	DEC eDX	DEC eBX	DEC eSP	DEC eBP	DEC eSI	DEC eDI
5xh	POP rAX	POP rCX	POP rDX	POP rBX	POP rSP	POP rBP	POP rSI	POP rDI
6xh	PUSH Iz (80186+)	IMUL Gv,Ev,Iz (80186+)	PUSH Ib (80186+)	IMUL Gv,Ev,Ib (80186+)	INS Yb,DX (80186+)	INS Yz,DX (80186+)	OUTS DX,Xb (80186+)	OUTS DX,Xz (80186+)
7xh	JS Jb	JNS Jb	JP Jb	JNP Jb	JL Jb	JNL Jb	JLE Jb	JNLE Jb
8xh	MOV Eb,Gb	MOV Ev,Gv	MOV Gb,Eb	MOV Gv,Ev	MOV Mw,Sw MOV Rv,Sw	LEA Gv,M	MOV Sw,Mw MOV Sw,Rv	group #10
9xh	CBW (8088) CBW/CWDE (80386+)	CWD (8088) CWD/CDQ (80386+)	CALL Ap	WAIT FWAIT	PUSHF Fv	POPF Fv	SAHF	LAHF
Axh	TEST AL,Ib	TEST rAX,Iz	STOS Yb,AL	STOS Yv,rAX	LODS AL,Xb	LODS rAX,Xv	SCAS Yb,AL	SCAS Yv,rAX
Bxh	MOV rAX,Iv	MOV rCX,Iv	MOV rDX,Iv	MOV rBX,Iv	MOV rSP,Iv	MOV rBP,Iv	MOV rSI,Iv	MOV rDI,Iv
Cxh	ENTER Iw,Ib (80186+)	LEAVE (80186+)	RET far Iw	RET far	INT3	INT Ib	INTO	IRET
Dxh	ESC 0	ESC 1	ESC 2	ESC 3	ESC 4	ESC 5	ESC 6	ESC 7
Exh	CALL Jz	JMP Jz	JMP Ap	JMP Jb	IN AL,DX	IN eAX,DX	OUT DX,AL	OUT DX,eAX
Fxh	CLC	STC	CLI	STI	CLD	STD	group #4 INC/DEC	group #5 INC/DEC etc.

note: The opcodes marked with * are aliases to other opcodes.

Appendix A2

IA-32 architecture
opcode groups

mod R/M	xx000xxx	xx001xxx	xx010xxx	xx011xxx	xx100xxx	xx101xxx	xx110xxx	xx111xxx
group #1 (80..83h)	ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
group #2 (C0..C1h) (D0..D3h)	ROL	ROR	RCL	RCR	SHL	SHR	SAL*	SAR
group #3 (F6..F7h)	TEST Ib/Iz	TEST* Ib/Iz	NOT	NEG	MUL AL/rAX	IMUL AL/rAX	DIV AL/rAX	IDIV AL/rAX
group #4 (FEh)	INC Eb	DEC Eb						
group #5 (FFh)	INC Ev	DEC Ev	CALL Ev	CALL Mp	JMP Ev	JMP Mp	PUSH Ev	
group #6 (0Fh,00h)	SLDT Mw SLDT Gv	STR Mw STR Gv	LLDT Mw LLDT Gv	LTR Mw LTR Gv	VERR Mw VERR Gv	VERW Mw VERW Gv	JMPE Ev (IA-64)	
group #7 (0Fh,01h)	SGDT Ms	SIDT Ms MONITOR (C8h) MWAIT (C9h) (see CPUID)	LGDT Ms	LIDT Ms	SMSW Mw SMSW Gv		LMSW Mw LMSW Gv	INVLPG M (80486+)
group #8 (0Fh,BAh)					BT	BTS	BTR	BTC
group #9 (0Fh,C7h)		CMPXCHG Mq (see CPUID)						
group #10 (8Fh)	POP Ev							
group #11 (0Fh,B9h)	UD2	UD2	UD2	UD2	UD2	UD2	UD2	UD2
group #12 (C6h) (C7h)	MOV							
group #13 (0Fh,71h)			PSRLW PRq,Ib (MMX) (66h) PSRLW VRo,Ib (SSE2)		PSRAW PRq,Ib (MMX) (66h) PSRAW VRo,Ib (SSE2)		PSLLW PRq,Ib (MMX) (66h) PSLLW VRo,Ib (SSE2)	
group #14 (0Fh,72h)			PSRLD PRq,Ib (MMX) (66h) PSRLD		PSRAD PRq,Ib (MMX) (66h) PSRAD		PSLLD PRq,Ib (MMX) (66h) PSLLD	

			VRo,Ib (SSE2)		VRo,Ib (SSE2)		VRo,Ib (SSE2)	
group #15 (0Fh,73h)			PSRLQ PRq,Ib (MMX) (66h) PSRLQ VRo,Ib (SSE2)	(66h) PSRLDQ VRo,Ib (SSE2)			PSLLQ PRq,Ib (MMX) (66h) PSLLQ VRo,Ib (SSE2)	(66h) PSLLDQ VRo,Ib (SSE2)
group #16 (0Fh,AEh)	FXSAVE M512 (see CPUID)	FXRSTOR M512 (see CPUID)	LDMXCSR Md (SSE)	STMXCSR Md (SSE)		LFENCE (SSE2-MEM)	MFENCE (SSE2-MEM)	CLFLUSH M (see CPUID) SFENCE (SSE-MEM)
group #17 (0Fh,18h)	PREFETCH-NTA M (SSE-MEM)	PREFETCH-T0 M (SSE-MEM)	PREFETCH-T1 M (SSE-MEM)	PREFETCH-T2 M (SSE-MEM)	HINT_NOP M (P6+)	HINT_NOP M (P6+)	HINT_NOP M (P6+)	HINT_NOP M (P6+)

note: The opcodes marked with * are aliases to other opcodes.

Appendix A3

IA-32 architecture
32bit mod R/M byte

r8(/r)			AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)			AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
sreg			ES	CS	SS	DS	FS	GS	res.	res.
eee			CR0	CR1	CR2	CR3	CR4	CR5	CR6	CR7
eee			DR0	DR1	DR2	DR3	DR4	DR5	DR6	DR7
/digit (opcode)			0	1	2	3	4	5	6	7
reg=			000	001	010	011	100	101	110	111
effective address	mod	R/M	value of mod R/M byte (hex)							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[sib]		100	04	0C	14	1C	24	2C	34	3C
[sdword]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX+sbyte]	01	000	40	48	50	58	60	68	70	78
[ECX+sbyte]		001	41	49	51	59	61	69	71	79
[EDX+sbyte]		010	42	4A	52	5A	62	6A	72	7A
[EBX+sbyte]		011	43	4B	53	5B	63	6B	73	7B
[sib+sbyte]		100	44	4C	54	5C	64	6C	74	7C
[EBP+sbyte]		101	45	4D	55	5D	65	6D	75	7D
[ESI+sbyte]		110	46	4E	56	5E	66	6E	76	7E
[EDI+sbyte]		111	47	4F	57	5F	67	6F	77	7F
[EAX+sdword]	10	000	80	88	90	98	A0	A8	B0	B8
[ECX+sdword]		001	81	89	91	99	A1	A9	B1	B9
[EDX+sdword]		010	82	8A	92	9A	A2	AA	B2	BA
[EBX+sdword]		011	83	8B	93	9B	A3	AB	B3	BB
[sib+sdword]		100	84	8C	94	9C	A4	AC	B4	BC
[EBP+sdword]		101	85	8D	95	9D	A5	AD	B5	BD
[ESI+sdword]		110	86	8E	96	9E	A6	AE	B6	BE
[EDI+sdword]		111	87	8F	97	9F	A7	AF	B7	BF
AL/AX/EAX/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
CL/CX/ECX/MM1/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
DL/DX/EDX/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
BL/BX/EBX/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
AH/SP/ESP/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
CH/BP/EBP/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
DH/SI/ESI/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
BH/DI/EDI/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF